Electronic Notes in Theoretical Computer Science 25 (2000)
URL: http://www.elsevier.nl/locate/entcs/volume25.html ?? pages

Real-Time Systems Development with MASS ??

Vered Gafni

Computer Science Department, Tel-Aviv University, Tel-Aviv, Israel
and MBT, Israel Aircraft Industries, Yehud, Israel

Yishai Feldman

School of Computer and Media Sciences,
The Interdisciplinary Center, Herzliya, Israel

Amiram Yehudai

Computer Science Department, Tel-Aviv University, Tel-Aviv, Israel

Abstract

In this paper, we demonstrate the capability of MASS, a real-time design language, for large systems specification. The paper presents a hierarchical specification of an automatic cruise controller that evolves through stepwise refinement. In particular, we show modular design, the separation of the functional and reactive concerns, and the succinct and intuitive nature of specifications in MASS.

1 Introduction

A real-time system consists of a plant where dynamic processes take place, and a controller (an embedded computer) aimed at the stabilization of the on-going processes at a required state. The controller design is especially complex, as compared with non-real-time applications, due to the reactive aspect of the its operation. This aspect comprises the need to synchronize its computations with the occurrences of the plant events (indicated by sensor data) and to accomplish their executions within hard deadlines determined by the controlled process dynamics (typical applications handle a large number of asynchronous events, thus giving rise to many intricate scenarios). On the

[1] Supported in part by grants from the Israel Science Foundation, the German–Israeli Foundation for Scientific Research and Development (GIF), and the Israel Ministry of Science.

other hand, real-time systems are usually safety-critical, and therefore their correctness is of an essential importance.

mass, the real-time design language employed in this paper, is based on a specification approach that handles the inherent complexity of real-time systems by completely separating the concerns of the functional and the reactive aspects in their specification. However, the underlying model formally relates these aspects in a way that enables comprehensive reasoning about the system behavior.

The key idea of the separation paradigm is to represent events and functions as different aspects of a single entity, called a *task*. In the functional view, a task denotes a computation that does not synchronize during execution with its environment (other tasks, or external systems). In the reactive view, a task is considered as a basic event that occurs at every termination of the task computation. Event expressions (constructed from basic events) are used to specify the activation of the tasks computations. Hence, due to the identification of events with task terminations, the specification of the computation of each task (given in the functional part) is completely independent of the reactive behavior in which it is employed.

mass [2] employs a single specification construct, called a *reaction*, that expresses an activation requirement for a task's computation in response to events (namely, terminations of tasks' computations). For instance, the requirement "every occurrence of p triggers the computation associated with q that must terminate within 5 time units" is succinctly expressed in mass by the reaction: $[p \Rightarrow q] \leq 5$. mass also provides a unique mechanism that enables hierarchical specification of large systems by stepwise refinement

A mass specification is made executable by a translation into a regular expression over signals denoting task activations and terminations. Operationally, we construct an automaton that operates synchronously to monitor the execution of asynchronous computations. A compiler, a run-time system, and simulation package, constructed for mass, enable its practical application to systems development Indeed, numerous large-scale system have been specified and implemented using mass [**?**,**?**].

mass is part of a formal specification framework for real-time systems which also includes the real-time logic plot, and verification procedures that check the correctness of a design in mass with respect to system requirements expressed in plot. In a nutshell, plot is an interval temporal logic [**?**] that allows the explicit expression of durations and timed occurrences of events. The logic is novel in its notion of causality, which is treated as a primitive semantic object. plot is decidable, and is associated with a deductive proof system. For the purpose of formal verification, a reaction is also interpreted by a plot formula that is consistent with the operational interpretation in the

[2] mass is acronym of 'Marionettes Activation Scheme Specification', a metaphor suggesting the separation of the activation mechanism from the activated puppets.

sense that it defines the same set of runs.

The rest of the paper is organized as follows. Section ?? provides an overview of mass. Section ?? presents a worked-out example of a large system specification with mass, and Section ?? surveys related work.

2 Overview of MASS

The specification unit in mass, called an *act*, presents the reactive behavior of a real-time system in the following form (boldfaced tokens are reserved words, and the notation $t\ldots$ means a finite list of terms of the type t).

Act *name* **is**

Tasks

system *task*, ...

environment *task*, ...

Reactions *reaction* ...

TimeBase *unit*

End

The **Tasks** section presents the tasks that comprise the real-time system, classified into the **environment** and **system** types. Environment tasks represent plant activities that are observable by the controller, but are not under its control (recall that in the mass model an activity is observed only by sensing the terminations of its executions). In contrast, the behavior of system tasks is fully controlled by the controller as specified by the reactions. The Reactions section specifies the activation requirements for the system tasks, and the Timebase section defines the sampling rate of the controller operation.

Tasks are declared in a functional notation of the form: $\mathsf{task\text{-}id} : \mathsf{Input} \rightarrow \mathsf{Output}$ where task-id is a name, and Input and Output are finite domains. We use the term void to denote a singleton. However, declarations of the form $\mathsf{q}{:}\mathsf{void}{\rightarrow}\mathsf{Output}$ and $\mathsf{q}{:}\mathsf{void}{\rightarrow}\mathsf{void}$ are usually abbreviated to $\mathsf{q}{:}{\rightarrow}\mathsf{Out}$ and q, respectively. Henceforth, we assume all tasks are declared with a void input domain, as indeed it turns to be the case in the example presented in this paper.

Every task is associated with a set of basic events, $q = v$ where v ranges over the output domain (in case of a task q – void output domain – the only basic event is also denoted by q). Each occurrence of an event $q = v$ denotes the termination of an execution of q that returned the output value v.

The events with respect to a given act are the basic events derived from the act tasks, the time events: $\mathrm{startup}, 0, 1, *$, and every event expression of the form: $\neg\alpha,\ \alpha \vee \beta,\ \alpha;\beta,\ \alpha \rightsquigarrow q$ where α, β are events, and q is a task.

The semantic domain for mass is a real-time systems model represented by timed-state sequences. We assume that basic events (the terminations of task

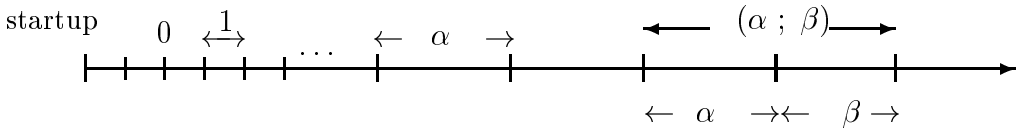

Fig. 1. Illustration of MASS events

executions) are observable along a discrete time axis (modeled by $\mathbb{N}$). Every task q is associated with a set of events C_q considered its causes. A *trace* of q is a function $tr_q : \mathbb{N} \to 2^{(C_q \times \mathbb{N})}$. Each element $(\alpha, i) \in tr_q(t)$ indicates an execution of q that terminated at t and was activated due to an occurrence of the cause α at the time instant $t - i$ (a trace value $tr_q(t) = \emptyset$ means that no execution of q terminated at t).

A trace represents a possible behavior of a task. A behavior of the entire system, called a *run*, consists of a trace for every task. Note that fixing the values of a trace as sets means that we allow concurrent executions of the task's computation, with the possibility that some of them terminate simultaneously (they are distinguished, however, by the causes and the activation times).

Events are interpreted over closed time intervals (including singletons $[n, n]$ representing time instants) with respect to a given run. Informally (see Fig. **??**):

- A basic event occurs at every time instant t such that $tr_q(t) \neq \emptyset$.
- $\alpha \rightsquigarrow q$ occurs together with those occurrences of q that were caused by α (namely, at the time instants t such that there exists an element $(\alpha, i) \in tr_q(t)$).
- *startup* occurs only at the instant the system starts operating, 0 occurs at every time instant, 1 occurs at every unit interval $[n, n+1]$, and $*$ occurs on every time interval (specifying an arbitrary duration).
- The logical symbols $\neg, \vee$ are denote negation, and disjunction, respectively.
- The symbol ; is the standard chop operator of interval temporal logic. The event $\alpha;\beta$ occurs at any interval composed of an occurrence of α immediately followed by an occurrence of β.

The standard temporal operator "eventually" is defined by $\Diamond\alpha \stackrel{\text{def}}{=} (*;\alpha;*)$, its dual "always" by $\Box\alpha \stackrel{\text{def}}{=} \neg\Diamond\neg\alpha$, and the "next" operator by $\bigcirc\alpha \stackrel{\text{def}}{=} 1;\alpha$.

A reaction describes an activation requirement for a system task by an expression of the form:

$$[\textit{activating-event} \Rightarrow \textit{response-task}] : \textit{aborting-event} \leq \textit{deadline}.$$

The activating and aborting events are normal event expressions, the response-task is the name of a system task declared in the act, and the deadline is a time expression. The activating event must be explicitly specified in a reaction, and is considered a cause of the response task in the real-time model for MASS events. In contrast, the aborting event and the deadline are both optional.

The intended meaning of a reaction is that the response task has to be activated following each occurrence of the activating event, and the termination of its execution must be observed within the duration designated by the deadline. However, MASS tasks are not executed instantaneously and therefore the termination of the response task cannot be observed but strictly after the occurrence of the activating event. A task terminates normally if its execution is completed no later than the first occurrence of the aborting event. Otherwise, the task execution is aborted at the occurrence of the aborting event, returning the value "!". If an execution exceeds the deadline, the system fails (the actual implementation of a system failure is left as a design decision).

For example, consider the reaction [TrainOut $\Rightarrow$ GateOpen] :TrainIn $\leq$10sec where TrainIn and TrainOut are environment tasks that indicate, respectively, the entrance and exit of a train in a railroad crossing, and the task GateOpen denotes the function that moves the gate up. The meaning of this reaction is that upon each occurrence of the event TrainOut, the task GateOpen should be activated, and its execution must be completed within 10 seconds. However, the execution of GateOpen is aborted, generating the event GateOpen=!, if the event TrainIn occurs while the gate is opening.

TimeBase.

The TimeBase declaration specifies a time unit which is used as a concrete measure for the interpretation of the event "1". Operationally, this quantity defines the sampling rate of the synchronous execution environment (see below). Thus, it determines the resolution at which events can be observed, and therefore it affects the extent to which activation requirements can be satisfied.

Execution environment.

The formal semantics of an act is expressed by a regular expression over runs. In practice, the regular expression is transformed into a finite automaton that monitors the occurrences of events, and reacts synchronously by activation and abortion of system tasks. The entire execution environment consists of a reactive executive and a functions executive that run concurrently the act-automaton and the execution of the activated functions, respectively.

The reactive executive runs synchronously at the time-base rate. At each time instant t_i, the input to the automaton is an observation set that consists of those tasks whose executions terminated at the period $[t_{i-1}, t_i)$. Terminations of environment tasks are reported by the plant sensors. For system tasks, the termination is reported by the functions executive. The automaton evaluates the observation set in order to identify occurrences of activating and aborting events specified by the act's reactions, and respectively generates activation and abortion commands for the functions executive.

Virtual events.

MASS contains an additional task type called *virtual* (also declared within the Tasks section). The basic events of a virtual task are identified with occurrences of events generated by other, previously defined, tasks. A virtual task has no executions of its own; its behavior merely reflects the executions of the tasks used to define its basic events. Thus, a virtual task can be used only in specifications of activating and aborting events. The basic form of a virtual task declaration is v **at** α where v is the virtual task name and α is a MASS event (called the marking event).

A virtual event is defined to occur at the time instants that end the intervals designating the occurrences of its marking event. A marking event may itself be defined in terms of virtual events (cyclic definitions are eliminated at compile time).

Virtual tasks are a useful means of abstraction, as they reduce a complex event expression to a basic event. In the general case, recursion enables the representation of regular expressions. For instance, a periodic event that occurs every 100ms can be defined as follows.

$$\text{Every100ms at startup} \vee (\text{Every100ms;100ms})$$

2.1 Modularity: Refinement, Composition, and Plays

Usually, the design of a system evolves through a number of abstraction levels, each of which adds design decisions that concretize the implementation towards a machine-executable program. MASS provides two mechanisms that enable a hierarchical modular representation of the controller design by a structure of acts.

- Task *refinement* associates a task with an act that is considered its implementation (concrete examples are given in the specification of the cruise controller in the next section). Operationally, each activation of the refined task causes a separate execution of the refinement.
- A *Composition* is a representation of a task q by a of acts $A_1, \ldots, A_k$, expressed as:

  $$\textbf{Act } q \textbf{ is } \quad (A_1 \| \ldots \| A_k) \quad \textbf{End}$$

  Operationally, the acts A_i become active simultaneously with each activation of q. A system task declared in one act may be declared as an environment task in any of the other acts, enabling these acts to react to the events generated by that task.

Task refinement and composition enable a hierarchical modular construction of a system. The process starts with an act specifying the operation of the system at a top-level view. Then, system tasks are separately refined by (a composition of) acts that elaborate their operations in terms of lower level tasks. The process can be iteratively applied to tasks in lower levels until all system tasks are represented by functional computations. A set of acts that

establish a hierarchical structure of refinements is called a *play*.

3 Specifying Large Systems with MASS

In this section, we demonstrate the applicability of MASS to the specification of large real-time systems. We present the specification of a cruise-control system that evolves through iterative hierarchical system refinement. This example is extensively worked out in the literature to demonstrate real-time specification frameworks (see Shaw [**?**] for survey). Thus, it is possible to compare MASS with other approaches.

3.1 Automatic Cruise Control

The Automatic Cruise Control (ACC) is intended to control the speed of a car according to the driver's instructions. The interface with the driver consists of a master switch (on/off), a 3-state speed-command lever (decrease, maintain, increase), a resume button, and the gas and brake pedals. The ACC takes over the speed control whenever the master switch is turned on, provided the car engine is working. The control is released either if the engine goes off, or the master switch is turned off.

While active, the ACC operates to maintain the car speed. The driver may instruct the system to increase or decrease the maintained speed by holding the speed-command lever at the corresponding position until the required speed is attained. The control operation is immediately suspended in case either the brake or the gas pedal are pressed. In this case, the driver may return control to the ACC by pressing the resume button (in which case the ACC returns to maintain the suspended speed, provided the brake and gas pedals are not pressed).

3.2 Hierarchical System Refinement

The specification of the ACC system is developed through iterative hierarchical system refinement. This method suggests to start a large system specification by refining the main thread of the system behavior. Everything else, though known to be part of the system, is considered part of the environment. In succeeding iterations, the structure is broadened by transferring environment activities into the system, and refining their behaviors. At any stage of the development the specification is amenable to simulation where the unspecified behavior is considered to be part of the environment.

In a development step, a task is refined either into a composition of tasks that represents a partition into concurrent activities, or by an act that specifies its design in terms of lower-level activities, also represented by tasks.

In the case of refinement by composition, we can independently proceed with any one of the constituent tasks. However, the partition of a composition need not specify all the constituents in order to be able to proceed with a

refinement of a certain task in the composition. This is possible, since in an act specification we make no assumption regarding the source of an environment task; whether it is external to the whole system, or a system task in another act of the play. Thus one may declare environment tasks, known to be part of another subsystem, even though the corresponding act has not been specified yet. Similarly, in case of task refinement by an act, one may proceed with the refinement of any of the system tasks that constitute the act independently of the refinement of other system tasks.

A refinement iteration may be taken to any desired extent, yielding a complete specification in the sense that no assumptions are made regarding unrefined tasks. In further iterations we go over each specification level, and either expand unrefined system tasks, or extend a composition with new acts that specify the behavior of tasks that were, until now, considered part of the environment. We proceed with that process until all system tasks are decomposed into basic components.

In the following subsections, this method is illustrated by working out the specification of the ACC.

3.3 Main Tread Specification

We start with the specification of what seems to us to be the main thread of the system operation (Fig. **??**). At the top level, the act ACC-Control controls the activation of the cruise-control function, denoted by the task CruiseCrl, according to the driver's instructions and the state of the engine.

```
Act ACC-Control is
  Tasks
    environment  Engine:→{off, on},    DriverCmd:→{start, stop}
    system       CruiseCrl
    virtual      EngineOn  at  (Engine=on;¬◇Engine=off)
  Reactions
    [ DriverCmd=start ∧ EngineOn  ⇒  CruiseCrl ]
                 : Engine=off∨DriverCmd=stop
End
```

In this act, the driver commands and the engine status are represented by the environment tasks DriverCmd and Engine, respectively. The reaction specifies CruiseCrl to be activated whenever the driver instructs the system to take over, provided the engine is turned on (denoted by the virtual task EngineOn). The task is aborted either due to a corresponding driver command, or when the engine is turned off (CruiseCrl is designed as a non-self-terminating operation, thus it must be externally aborted in order to be stopped).

The task EngineOn demonstrates a typical usage of virtual events, as phase designators. This declaration sustains the fact that the engine is turned on by generating the event EngineOn repeatedly at every time instant as long as the engine remains in this phase.

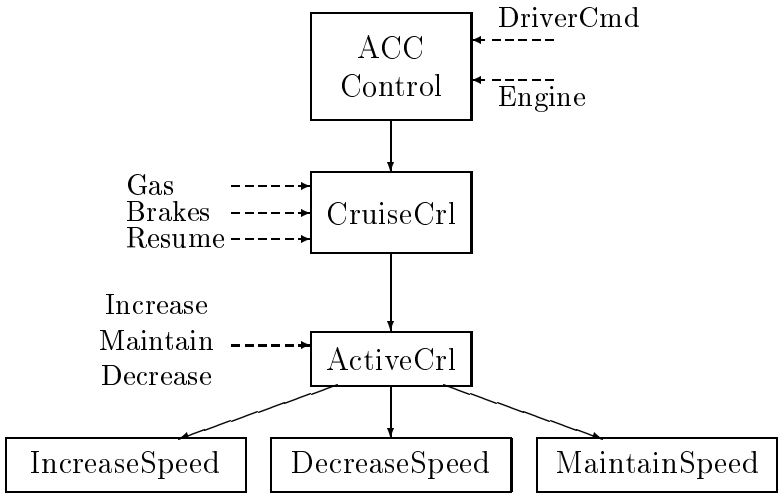

Fig. 2. Main control-thread of ACC

Note that at this stage we are not concerned with how the driver commands and the engine status are monitored, therefore they are represented by environment tasks. At a later stage, we can independently (in different acts) specify the activation requirements for these operations (as indeed we do), and compose them with the control operation. Also note that the time domain is not specified. It is expected to be added at a later stage after a timing analysis of the activation requirements.

The task CruiseCrl is further refined as follows.

```
Act CruiseCrl
  Tasks
    environment Brakes,  Gas,  Resume
    system  ActiveCrl
    virtual
      Suspend at  (CruiseCmd;first(Brakes ∨ Gas)),
      CruiseCmd  at  (startup ∨ ((Suspend;first(Resume∧¬Brakes))
  Reactions   [ CruiseCmd ⇒ ActiveCrl ] :Suspend
End
```

first(α) is defined by $(*;\alpha) \wedge \neg\Diamond(\alpha;1)$, indicating an interval that ends with the first occurrence of α.

The virtual task Suspend designates the transition from active to suspended control (due to gas or brake pedal press, represented by the environment tasks Brakes and Gas). The virtual task CruiseCmd designates the transition to active control, either at startup or due to a press of the resume button (represented by the environment task Resume) in a suspended state while the brakes are

not pressed. The task ActiveCrl denotes the ACC active control operation.

Next we refine ActiveCrl as follows.

```
Act ActiveCrl
  Tasks
    environment  Decrease,  Maintain,  Increase
    system  MaintainSpeed,  DecreaseSpeed,  IncreaseSpeed
  Reactions
    [ (startup ∨ Maintain) ⇒ MaintainSpeed ] :Decrease ∨ Increase
    [ Decrease ⇒ DecreaseSpeed ] :Maintain
    [ Increase ⇒ IncreaseSpeed ] :Maintain
End
```

The environment tasks Increase, Decrease, and Maintain designate the corresponding changes in the lever position, and the task CurrentSpeed provides the car speed at every time instant (it is the same task as declared in CruiseCrl). The system tasks MaintainSpeed, DecreaseSpeed, IncreaseSpeed implement the control operations corresponding to the current lever position.

The concrete behavior of the control operations is given by the refinements below. The act MaintainSpeed operates a control-loop, at 10Hz rate, [3] to maintain the car speed as recorded at the act activation (the first reaction).

```
Act MaintainSpeed
  Tasks
    system  SpeedControl,  ReadSpeed
  Reactions
    [ startup ⇒ ReadSpeed ] ≤70ms
    [ Every100ms ⇒ SpeedControl ] ≤20ms
End
```

The acts, DecreaseSpeed and IncreaseSpeed operate to increase and decrease the car speed by moving the throttle up and down, respectively, in an open loop.

```
Act DecreaseSpeed
  Tasks
    system  RetractThrottle
  Reactions
    [ Every100ms ⇒ RetractThrottle ] ≤20ms
End

Act IncreaseSpeed
  Tasks
```

[3] Events designating periodic signals are not explicitly declared in acts. They are assumed to be pre-defined by virtual tasks. For instance, a 1Hz signal is defined by Every1sec at startup ∨ (Every1sec;1sec).

```
    system    AdvanceThrottle
  Reactions
      Every100ms ⇒ AdvanceThrottle ] ≤20ms
End
```

The last acts deal with basic activation requirements (namely, the system tasks can not be further refined). Hence, the main thread of the system refinement has been completed. In the next section, we describe the next iteration where the play is completed with horizontal (composition) acts.

3.4 Completion of the Play

At this stage, after the main thread of the system operation had been specified, we elaborate declarations of environment tasks, and combine them into the entire system specification. At the top level of the specification, we define the acts EngineMon, DriverCmdMon, and SpeedMon.

The act DriverCmdMon encapsulates the master-switch that turns the ACC on and off. This act is responsible for generating the events denoted by DriverCmd (used by the act ACC-Control).

```
Act DriverCmdMon
  Tasks
    environment MasterSwOff, MasterSwOn
    virtual DriverCmd:→{ start, stop }
          where DriverCmd=start   at   (0.2sec;MasterSwOn)
                DriverCmd=stop    at   MasterSwOff
End
```

The environment tasks MasterSwOff and MasterSwOn represent hardware interrupts generated upon turning the master switch off and on, respectively. DriverCmd is declared as a virtual task considered an abstraction of the interrupts. Note that every activation commands that occur within the first 0.2 seconds of the system operation are ignored. This delay, although not specified in the ACC requirements, is necessary in order to allow speed computation and engine monitoring prior to entering the active-control state.

This act is superfluous since we could directly use MasterSwOff and MasterSwOn in ACC-Control. However, in the context of a whole system development and maintenance, it is good design practice to hide the specific implementation of the sensor. For instance, in a later stage of the development, we could decide to modify the implementation by using polling instead of interrupts (see below). With the suggested design, this would not affect the acts that use the task DriverCmd.

The act EngineMon monitors the engine in order to detect the events indicating the engine being turned on and off (represented by the task Engine which is used by the act ACC-Control).

```
Act EngineMon
  Tasks
    system EngineState:→{ off, on }
    virtual Engine:→{ off, on }
          where Engine=off  at
                    Startup ∨ (Engine=on;first(EngineState=off))
                Engine=on  at
                    (Engine=off;first(EngineState=on)
  Reactions [ Every20ms ⇒ EngineState ] < 10ms
End
```

Here, we prefer an implementation by polling (mentioned above). The task EngineState periodically samples the state of the engine, and Engine is declared as a virtual task that designates the changes in the state of the engine operation.

In order to compose the additional acts with ACC-Control we add an upper level act, ACC, declared as follows.

```
Act ACC is  ( ACC-Control || DriverCmdMon || EngineMon )  End.
```

At the next level, we define the acts BrakesMon, GasMon and ResumeMon that encapsulate, respectively, the events Brakes, Gas and Resume. We assume that these events are generated by interrupts, hence the acts are very simple (similar to DriverCmdMon).

However, the composition of these acts into the play turns out to be somewhat tedious. Normally, we would need to rename the act CruiseCrl, say by MainCruiseCrl, and then introduce the definition

```
Act CruiseCrl is  ( MainCruiseCrl || BrakesMon || GasMon || ResumeMon )
                                                          End.
```

This procedure is undesirable for a number of reasons.

- It enforces modifications of already existing part of the specification.
- It inflates the play with superfluous hierarchical levels (not contributing levels of essential information).
- It presents at the same abstraction level acts that are not equally significant in the specification of the system behavior at this level. Such a presentation seems unnatural, and blurs the picture.

Therefore, we allow naming a composition by one of the constituent act. This form emphasizes the subsystem which seems to be central (in the developer's view), and does not force a designer to have a complete view of the system structure at the initial stage.

Thus, in our case, we define the composition

```
Act CruiseCrl is  ( CruiseCrl || BrakesMon || GasMon || ResumeMon )  End.
```

Finally, we define the act LeverMon that generates the events Decrease, Maintain, and Increase, which indicate the corresponding transitions in the lever positions. The implementation of this act relies on periodic sampling of the lever and identification of the state transitions by virtual tasks.

```
Act LeverMon
  Tasks
    system LeverPosition:→{ down, mid, up }
    virtual
      Maintain  at  startup ∨ ((Increase ∨ Decrease);first(LeverPosition=mid)),
      Increase  at  (Maintain;first(LeverPosition=up)),
      Decrease  at  (Maintain;first(LeverPosition=down))
  Reactions [ Every200ms ⇒ LeverPosition ] < 10ms
End
```

Here, again, we employ the relaxed form of composition, and define

Act ActiveCrl is (ActiveCrl || LeverMon) End.

4 Related Work and Discussion

MASS is a synchronous language that monitors the execution of asynchronous computations under real-time constraints. This execution paradigm, which overcomes the essential limitation of infinitesimally short computations assumed by the synchronous model [?], is enabled by the idea of identifying system events with the terminations of the computations.

CRP introduced by Berry et al. [?] presents an alternative approach. The language extends ESTEREL [?] with a special construct, exec L:P, which implicitly associates the label L with the events (signals) denoting the start, termination, and abortion of an execution of P. This approach is inherently restricted since once engaged with an execution, a program cannot respond to additional activations. For instance, if P is an asynchronous computation, a requirement like "activate P upon every occurrence of the event α" is not expressible in this formalism.

Synchronous Eifel(previously called *Embedded Eifel* [?]) employs an execution environment similar to MASS, but without monitoring asynchronous computations. The reactive behavior is specified in the synchronous language ESTEREL, augmented with a special construct schedule(f) where f is a function. Functions scheduled this way, called background services, are run asynchronously (in a non-preemptive manner) in the time slots between the termination of the synchronous computations and the next time instant, with no deadlines. A background service can communicate with the reactive part by a special command that adds a signal to the next time instant.

MASS is an executable language in the full operational sense. A similar approach is also taken in the design of SAFE [?], which is a procedural real-

time programming language with interval temporal-logic semantics (however, SAFE does not support concurrency).

5 Conclusion

The main contribution of the plot/mass framework is the idea of associating each primitive event in a real-time system with a concrete function, and interpreting every termination of the function execution as an occurrence of the event. This idea gives rise to a real-time specification framework that enables the design of real-time systems, and formal reasoning about the behavior of asynchronous functions whose executions are controlled synchronously.

mass, introduces a new declarative activation-oriented specification approach, which we believe is natural for real-time system representation, and easy for system designers to understand and use. Another novelty of mass is the concept of activation refinement, which provides an essential means for modular development and reasoning.

Finally, the fact that events and computations are semantically related provides for a complete separation of the specification of the reactive and algorithmic aspects of a system, an important software-engineering concern.